JavaScript notes

1. Adding internal JavaScript to HTML

a. On page script

<script type="text/javascript"> //JS code goes here </script>

b. External js file

<script src="file.jst"> </script>

2. Javascript Basics

a. Datatype

i) string, number, boolean, null , object

ii) null is explicitly assigned, it means intentionally don't want to assign no value or should be empty.

iii)
On the other hand, undefined means that the variable or object has not been declared, or has not been given a value.It is automatically assigned.

iv) typeOf(x) :- determine the datatype

b. Variables

In JavaScript, let, const, and var are used for declaring variables, but they differ in terms of their scope, hoisting behavior, and reassignment abilities. Here are some examples to illustrate their differences:

i) **Le**t :- block scope, use if value can be change

→ block-scoped variables (block of code within bracket). This means that variables declared with let are only accessible within the block in which they are declared.
→ can reassign value.

ii) **var:**

→ var is the oldest way to declare variables in JavaScript. Variables declared with var are function-scoped, meaning

they are accessible within the function in which they are declared. If var is declared outside of a function, it is global, meaning it can be accessed from anywhere in the code.

→ value can be reassigned.

iii) Const:

- block scope
- not reassignable
- not redeclarable

Example

→ var is a function scope ***

```
if(true){
    var varVariable = 'This is var';
    var varVariable = 'This is var again';
}

console.log(varVariable);  → This is var again
```

→ let is a block scope ***

```
if(true){
    let letVariable = 'This is let';
    let letVariable = 'This is let again';
```

→ let variable can't re-define but we can re-assign value

```
    console.log(letVariable); → let letVariable = 'This is let again';^SyntaxError:
                                                    Identifier 'letVariable' has
                                                    already been declared
}

console.log(letVariable); → ReferenceError: letVariable is not defined
```

→ const variable can't re-define and re-assign value
→ const is a block scope ***

```
if(true){
   const constVarible = {
      name: 'JavaScript',
      age: '25 years',
   };
   constVarible.name = 'JS';

   console.log(constVarible) → {name: 'JS', age: '25 years'} <= we can update
                                const variable declared object
}
```

c. **Operator**: ==, ====, +, -, +=, -=, !=,..

→ == it convert the variable values to the same data type before performing comparison.

→=== it does not convert data type of variable before performing comparison.

d. Strings and its methods

```
Var abc = "abcdefghijklmnopqrstuvwxyz";
var esc = 'i don\'t \n know';     // \n new line
var len = abc. Length;            // string length
abc. Indexof("lmno");     // find substring, -1 if doesn't contain
abc. Lastindexof("lmno");       // last occurance
abc. Slice(3, 6); // extract part of string from 3 including
                        upto excluding 6
abc. Replace("abc", "123"); // find and replace, takes
                                regular expressions
abc. Touppercase();     // convert to upper case
abc. Tolowercase();     // convert to lower case
abc. Concat(" ", str2);    // abc + " " + str2
abc. Charat(2);           // character at index: "c"
```

```
abc[2];                 // unsafe, abc[2] = "c" doesn't work
abc. Charcodeat(2);     // character code at index: "c" -> 99
abc. Split(", ");        // splitting a string on commas gives an array
abc. Split("");         // splitting on characters
128. Tostring(16);      // number to hex(16), octal (8) or binary (2)
abc.trimStart()
abc.trimEnd()
```

e. Functions:

- **High order function:-** In simple words, a higher-order function is a function that receives a function as an argument or returns the function as output.

- **Example**
  filter(), map(), reduce().

- ```
  function myFunction(a,b) {
    let carName = "Volvo"; // local
    variable return a* b;
  }
  Let x = myFunction(10,20);
  // need to know local, global variables, argument,
  parameter, call(), apply(), bind()
  ```

f. Conditional statement

i) **if else if**

```
If (condition1) {
        // block of code to be executed if condition1 is true
} else if (condition2) {
        // block of code to be executed if the condition1 is
        false and condition2 is true
} else {
        // block of code to be executed if the condition1 is
        false and condition2 is false
}
```

ii) **switch statement**

```
Switch
        (expression)
        { case x:
                // code block
                break;
        case y:
                // code block
                break;
        default:
```
```

```
        // code block
    }
```

g. Iterative Statement

i) for loop

```
For (initialization; condition; incrementation; ) {
    // code block to be executed
}
```

Example

```
for (let i = 0; i < 5; i++) {
    text += "iteration number: " + i + "<br>";
}
```

ii) while loop

```
while (condition) {
    // code block to be executed
}
```

iii) do while loop

```
do {
  // run this code in
  block i++;
} while (condition);
```

h. Array

→ An array is a special type of object that stores an ordered list of elements. The elements can be of any data type, and are accessed using an index that starts at 0.

```
var fruit = ["element1", "element2", "element3"];
```

methods

i) concat() //Joins two or more arrays together.

ii) indexOf() //Returns the index of the specified item from the

array.

iii) join() //Converts the array elements to a string.

iv) pop(), push() //Deletes the last element of the array.

v)reverse() //This method reverses the order of the array elements.

vi)sort() //Sorts the array elements in a specified manner.

viii)shift, unshift() // method removes the first array element and "shifts" all other elements to a lower index, adds a new element to an array (at the beginning), and "unshifts" older elements.

ix)splice(where_index, removeItem_index,"new element")

//add new item in where_index, remove item in removeItem_index and add element also return deleted element with modified array

x)slice() //same as mentioned in string section return new array upon slicing array from certain index

xi) array.flat

i. ~~Objects~~

→On the other hand, an object is a collection of key-value pairs, where each key is a string that acts as an identifier for its corresponding value. The values can be of any data type, including arrays and other objects.

i) Ways to create the object

- **literals like**,

```
const obj = {
        x:50,
        y:60
    };
```

- Using **constructor function** like

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

const john = new Person("John",
30);
```

- Using **class syntax**:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

const john = new Person("John", 30);
```

- Using **Object.create():**

```
const person = {

  name: "",
  age: 0,
  setPerson(name, age) {
    this.name = name;
    this.age = age;
  }
};

const john = Object.create(person);
john.setPerson("John", 30);
```

ii)<u>Method of Global object (inbuilt)</u>

- **Object.prototype:** It's an object that can be used to add properties and methods to all objects created from a constructor function.

- **Object.keys():** It's a method that returns an array of all the property names of an object.

```
const person = { name: 'John', age: 25, gender: 'male' };
const keys = Object.keys(person);
console.log(keys); // logs ["name", "age", "gender"]
```

- **Object.values():** It's a method that returns an array of all the property values of an object.

```
const person = {name: 'John', age: 25, gender: 'male' };
const values = Object.values(person);
console.log(values); // logs ["John", 25, "male"]
```

- **Object.entries():** It's a method that returns an array of all the key-value pairs of an object.

```
const person = {name: 'John', age: 25, gender: 'male' };
const entries = Object.entries(person);
console.log(entries); // logs [["name", "John"], ["age", 25], ["gender", "male"]]
```

- **Object.assign():** It's a method that is used to copy properties from one object to another.

```
const person = { name: 'John', age: 25 };
const moreInfo = { gender: 'male', occupation: 'engineer' };
const updatedPerson = Object.assign({}, person, moreInfo);
console.log(updatedPerson); // logs { name: 'John', age: 25, gender: 'male', occupation: 'engineer' }
```

- **Object.defineProperty():** It's a method that is used to add a new property to an object, or modify the attributes of an existing property.

```
const person = {};
Object.defineProperty(person, 'name', { value: 'John', writable: false });
console.log(person.name); // logs "John"
person.name = 'Mike';
console.log(person.name); // still logs "John"
```

- **Object.getOwnPropertyDescriptor():** It's a method that returns an object that describes the attributes of a given property.

- **Object.getOwnPropertyNames():** It's a method that returns an array of all property names (including non-enumerable properties) found directly on a given object.

- **Object.seal():** It's a method that prevents new properties from being added to an object and marks all existing properties as non-configurable.

```
const person = { name: 'John', age: 25 };
Object.seal(person);
```

```
person.gender = 'male';
console.log(person.gender); // logs "undefined"
person.name = 'Mike';
console.log(person.name); // logs "Mike"
delete person.age;
console.log(person.age); // logs 25
```

- **Object.freeze():** It's a method that prevents new properties from being added to an object, and marks all existing properties as non-writable and non-configurable.

```
const person = { name: 'John', age: 25 };
Object.freeze(person);
person.gender = 'male';
console.log(person.gender); // logs "undefined"
person.name = 'Mike';
console.log(person.name); // logs "John"
delete person.age;
console.log(person.age); // logs 25
```

iii) Methods vs Properties

→ Properties are the values associated with an object, and they describe the characteristics or attributes of the object. Examples of properties of an object include its name, age, and color.

→ Methods, on the other hand, are functions that are associated with an object and are used to perform some action or operation on the object. Examples of methods of an object include functions that perform calculations, update the properties of the object, or return information about the object.

Example of properties

```
const person = {
  name: 'John',
  age: 25,
  gender: 'male'
};
```

```
console.log(person.name); // Outputs 'John'
console.log(person.age); // Outputs 25
console.log(person.gender); // Outputs 'male'
```

Example of methods

```
const person = {
  name: 'John',
  age: 25,
  gender: 'male',
  greet: function() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
};

person.greet(); // Outputs 'Hello, my name is John and I am 25 years old.'
```

j. Classes, Inheritance and constructor

→ In JavaScript, classes are a relatively recent addition to the language (introduced in ES6), and they provide a way to **create objects that share common properties and methods**. **Classes allow** you to define blueprints for objects that can be instantiated using the new keyword.

Here's an example of a basic class definition in JavaScript:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}
```

```
const john = new Person('John', 25);

john.greet(); // Outputs 'Hello, my name is John and I am 25 years old.'
```

→ In this example, we define a Person class that has a **constructor** method (a constructor is a function that is used to create and initialize objects created with the 'new' keyword.), which is **called when a new instance** of the class is created using the **new** keyword. The constructor method **sets** the name and age properties of the object to the values passed in as arguments.

→ The Person class also has a greet method that outputs a greeting message using the name and age properties of the object. We create a new instance of the Person class named john and call the greet method on it.

→ Inheritance is a mechanism that allows you **to define a new class based on an existing class**. In JavaScript, you can achieve inheritance using the **extends** keyword. The new class is called the child class, and the existing class is called the parent class.

→ Here's an example that demonstrates how inheritance works in JavaScript:

```
class Student extends Person {

        constructor(name, age, major) {
                super(name, age);
                this.major = major;
        }

         study() {
                console.log(`I am studying ${this.major}.`);
        }
}
         const jane = new Student('Jane', 20, 'Computer Science');
         jane.greet(); // Outputs 'Hello, my name is Jane and I am 20
                              years old.'
         jane.study(); // Outputs 'I am studying Computer Science.'
```

→ In this example, we define a Student class that extends the **Person**

class using the **extends** keyword. The Student class has a constructor method that calls the **super** method to call the constructor of the parent class and set the name and age properties of the object. It also sets the major property of the object.

→ The Student class also has a study method that outputs a message about what the student is studying. We create a new instance of the Student class named jane and call the greet and study methods on it.

→ The super method is used to call the constructor of the parent class from the constructor of the child class. This is necessary because the name and age properties are defined in the parent class, and we need to call its constructor to initialize them.

→ The constructor method is optional in a child class. If it is not defined, the parent class constructor will be called automatically with no arguments. However, if the child class does define a constructor method, it must call the super method to call the parent class constructor.

→ **this and new ketword:**

→ new is used to create a new instance of an object, using a constructor function. When you call a function with new before it, JavaScript creates a new object and sets the this keyword to reference that new object. The new object is then returned by the constructor function, allowing you to access its properties and methods.

→ this is a keyword that refers to the current object being executed. In the context of a constructor function, this refers to the newly created object that the constructor is being called with.

3. Advance javascript

a) Array iteration

i) for …of

→for...of() is a loop used to iterate over an iterable object such as an **array, string**. It provides an easier syntax than the traditional for loop and eliminates the need for using an index variable.

→ **const arr = [1, 2, 3, 4, 5];**

```
for (const num of arr) {
  console.log(num);
}
// Output: 1 2 3 4 5
```

ii) forEach()

→ forEach() is another higher-order function that executes a provided function once for each element in an **array**. It does not create a new array, but can be used for performing some action on each element of an array.

→ **const arr = [1, 2, 3, 4, 5];**

```
arr.forEach(function(num) {
  console.log(num);
});
// Output: 1 2 3 4 5
```

iii) map()

→map() is a higher-order function that creates a new array by calling a function on each element of an existing array. It returns an array of the same length as the original array, with each element transformed by the function provided.

```
const arr = [1, 2, 3, 4, 5];
const squaredArr = arr.map(function(num) {
  return num * num;
});

console.log(squaredArr);
// Output: [1, 4, 9, 16, 25]
```

iv) **reduce()**

→ reduce() is a higher-order function that reduces an array to a single value by calling a provided function on each element of the array. The function takes two arguments, an accumulator and the current value, and returns the updated accumulator value. The final result is the accumulated value at the end of the iteration.

→ **syntax**

```
const result = array.reduce(function(accumulator,
                    currentValue, index, array) {
  // Return updated accumulator value
}, initialValue);
```

→ **Example**

```
const arr = [1, 2, 3, 4, 5];

const sum = arr.reduce(function(acc, num) {
  return acc + num;
}, 0);

console.log(sum);
// Output: 15
```

v) **filter()**

→ filter() is another higher-order function that creates a new array by filtering out elements of an existing array based on a condition specified by a function. It returns an array containing only the elements that meet the condition.

→ **Example**:

```
const arr = [1, 2, 3, 4, 5];
const evenArr = arr.filter(function(num) {
  return num % 2 === 0;
});
console.log(evenArr);
// Output: [2, 4]
```

vi) for..in

→ The for...in loop is used to iterate over the properties of an object. It iterates over all enumerable properties, including inherited properties, and the loop variable takes on the name of each property in turn. It should not be used to iterate over arrays, as it can result in unexpected behavior.

→
```
const obj = { a: 1, b: 2, c: 3 };

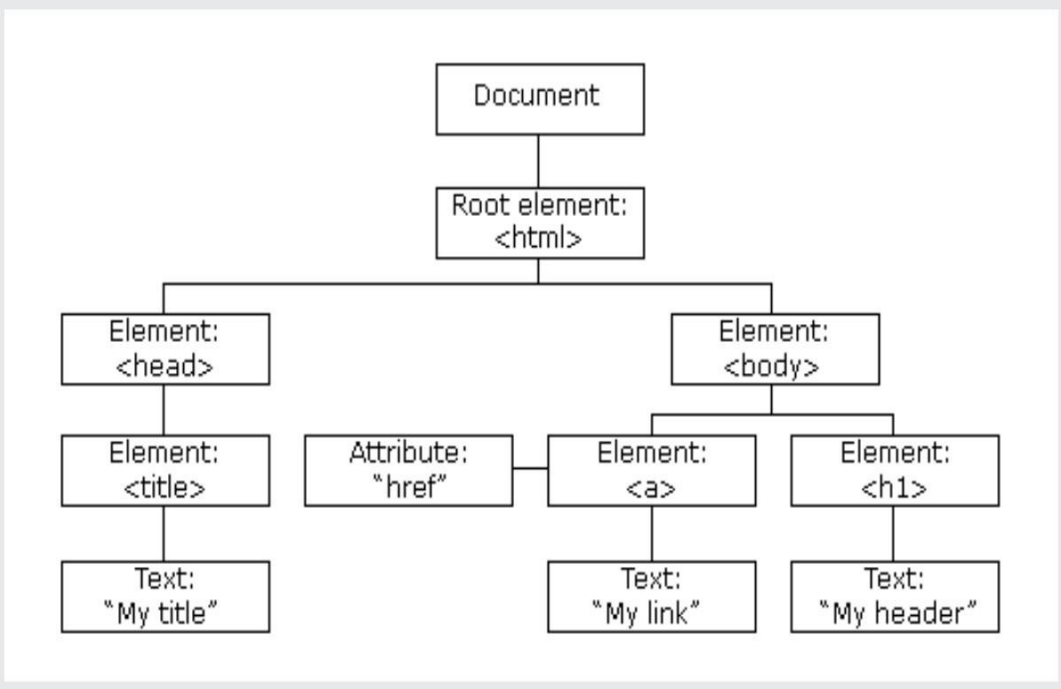
for (const prop in obj) {
  console.log(`${prop}: ${obj[prop]}`);
}
// Output: a: 1 b: 2 c: 3
```

vii) Remaining Iteration statement

- every() method check if all array values pass a test.
- some()method check if some array values pass a test.
- indexOf() method searches an array for an element value and returns its position.
- Array.lastIndexOf() is the same as Array.indexOf(), but returns the position of the last occurrence of the specified element.
- find() method returns the value of the first array element that passes a test function.

b) Document Object Manipulation

→ DOM manipulation is the process of changing the content, structure, or style of a web page using JavaScript. The DOM is an object-oriented representation of the HTML and XML documents that make up a web page, providing a hierarchical structure of nodes. With DOM manipulation, developers can use JavaScript to access and modify the properties and attributes of DOM elements dynamically, including changing text or HTML content, adding or removing attributes, manipulating the document tree by adding, moving, or removing elements, and changing the style or classes of an element.



i) Dom documents

c)

- **Finding html element**

```
Document.getElementById();
Document.getElementByTagName();
Document.getElementByClassName();
Document.querySelector(); // select the first matching
                          element
Document.querySelectorAll();// select all matching
ele…
```

- **Changing html element**

```
element.innerHtml = new html content
                    // change the inner html of an element
element.attribute = new value
                    //  change  the  attribute  value  of
                    html element
element.style.property = new style
                    // change the style of  html  an element
element.setAttribute(attribute, value)
                    // change the attribute value of
                    html element
```

- **Adding and deleting element**

```
document.createElement(element)
document.removeChild(element)
document.appendChild(element)
document.replaceChild(old, new)
document.write(text)
```

// there are more refer w3school , above mention is enough

d) JS Async

i) JS asynchronous:

→ JavaScript is a single-threaded programming language, which means that it can only execute one piece of code at a time. Asynchronous programming is a way to allow the program to continue running while waiting for a long-running task to complete, such as fetching data from a server or reading a file from disk.

→ In JavaScript, asynchronous programming is achieved through the use of callback functions, Promises, and async/await. When a long-running task is started, JavaScript does not block the thread waiting for the task to complete. Instead, it registers a callback function to be executed when the task is finished, and then continues executing the rest of the code.

→**Event Loop:**

→ The event loop is the mechanism that JavaScript uses to manage asynchronous tasks.

→It is a loop that continuously checks the event queue for new events to process.

→When an event is added to the queue, such as a timer finishing or a Promise being resolved, the event loop takes the callback function associated with that event and adds it to the call stack, which is a stack of functions that are currently being executed.

→ The event loop ensures that only one function is executed at a time, and that the functions are executed in the order in which they were added to the call stack.

→This prevents blocking of the thread and allows the program to continue running while waiting for asynchronous tasks to complete.

→In summary, asynchronous programming in JavaScript allows the program to continue running while waiting for long-running tasks to complete, and is achieved through the use of callback functions, Promises, and async/await. The event loop is the mechanism that manages asynchronous tasks and ensures that functions are executed in the correct order, preventing blocking of the thread.

ii) JS callback

→ In JavaScript, **a callback function is a function that is passed as an argument to another function and is invoked when a certain event occurs or a task is completed**. Callback functions allow for **asynchronous programming,** where code can continue executing while waiting for a response from an asynchronous task.

Here is an example of a callback function:

```
function myCallbackFunction() {
  console.log("Callback function has been invoked!");
```

```
}

function higherOrderFunction(callback) {
  console.log("Higher order function is executing...");
  callback();
}

higherOrderFunction(myCallbackFunction);
```

→ In this example, myCallbackFunction is a callback function that is passed as an argument to higherOrderFunction. When higherOrderFunction is executed, it logs a message and then invokes the callback function that was passed to it as an argument. The callback function, myCallbackFunction, simply logs a message to the console when it is invoked.

→ Callbacks are commonly used in JavaScript for handling asynchronous operations such as making API requests or responding to user events. By passing a callback function as an argument to an asynchronous function, the callback can be executed once the asynchronous operation is complete, allowing code to continue executing in the meantime.

→ In JavaScript, a callback function is passed as an argument to another function and is invoked when a certain event occurs or a task is completed. A higher-order function takes one or more functions as arguments or returns a function as its result. A callback function can be used as an argument to a higher-order function to perform a specific operation.

iii) JS Promises

→ A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows us to handle asynchronous operations in a more readable and manageable way, like **pending, resolved, and rejected.**

Example

```
new promise( (res, rej) => {
        settimeout(() => res(1), 1000);
}).then( (res) => {
```

```
        console. log(res); // 1
        return res*2
}).then( (res) => {
        console. log(res);  // 2
}).catch(( => {
       Console.log(err);
})
```

Another Example

```
var x = fetch(SOME_URL, SOME_POST_DATA)
   .then((response) => response.json())
   .then((responseJSON) => {
     // do stuff with responseJSON here...
     console.log(responseJSON);
   });
```

→ In this example, we create a new Promise using the Promise constructor, which takes a function as an argument. This function, called the executor function, takes two arguments: resolve and reject. Inside the executor function, we set a timeout of 1 second and call the resolve function with the value "Promise resolved!" when the timeout finishes.

→ We then call the then method on the Promise object, which takes a callback function that is executed when the Promise is resolved. In this case, we log the result to the console.

iv) JS Async/Await

→ Async/await is a way of writing asynchronous JavaScript code that is easier to read and understand compared to Promises and callbacks. It is built on top of Promises and uses the same underlying mechanisms, but provides a cleaner syntax for handling asynchronous operations.

→ Async/await allows us to write asynchronous code that looks and behaves like synchronous code. This is achieved by using the

async keyword to define an asynchronous function, and the await keyword to wait for the result of a Promise before moving on to the next line of code.

→Here is an example of using async/await to fetch data from an API:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
fetchData();
```

→fetch method, which returns a Promise that resolves to a Response object. We then use the await keyword again to wait for the result of the json method on the Response object, which also returns a Promise that resolves to the parsed JSON data.

e) JS Json(javascript object notation)

→JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It consists of key-value pairs, where the keys are strings and the values can be any valid JSON data type, including strings, numbers, booleans, arrays, and other objects. JSON is often used for data exchange between web servers and web applications and can be easily converted to and from JavaScript objects using the JSON.parse() and JSON.stringify() methods

i) JSON PARSE

→ JSON.parse() is a built-in JavaScript method that is used to parse a JSON string into a JavaScript

object. It takes a valid JSON string as input and returns a JavaScript object that corresponds to the JSON data. For example:

```
const jsonString = '{"name": "John Smith",
    "age": 30, "isMarried": true, "hobbies":
    ["reading", "traveling", "hiking"]}';
const data = JSON.parse(jsonString);
    console.log(data); // {name: "John Smith",
    age: 30, isMarried: true, hobbies: Array(3)}
```

ii) JSON STRINGIFY

→JSON.stringify() is a built-in JavaScript method that is used to convert a JavaScript object into a JSON string. It takes a JavaScript object as input and returns a JSON string that corresponds to the object. For example:

```
const data = {
  name: "John Smith",
  age: 30,
  isMarried: true,
  hobbies: ["reading", "traveling", "hiking"]
};
const jsonString = JSON.stringify(data);
console.log(jsonString); // '{"name":"John
Smith","age":30,"isMarried":true,"hobbies":["reading","traveli
ng","hiking"]}'
```

f) JSON Web API

i) Localstorage

→ localStorage is a built-in web browser feature in JavaScript that provides a way to store key-value pairs in a user's web browser. The stored data remains **persistent** even after the user closes and reopens the web page or browser, making it a useful tool for implementing client-side data caching and storage.

→The localStorage object has a simple API that consists of methods to set, retrieve, and remove data. The setItem() method is used to store data with a key, the getItem() method is used to retrieve data by key, and the removeItem() method is used to remove data by key.

```
// Store data in local storage
localStorage.setItem("username", "John");

// Retrieve data from local storage
const username = localStorage.getItem("username");
console.log(username); // "John"

// Remove data from local storage
localStorage.removeItem("username");
```

ii) **session storage:**

→The **sessionStorage** object is identical to the localStorage object.

→The difference is that the sessionStorage object stores data for one session.

→The data is deleted when the browser is closed.

→**sessionStorage**.getItem("name");

g) EXTRA

i) JS fromEntries(2019)

→The fromEntries() method creates an object from iterable key / value pairs.

```
const fruits = [
["apples", 300],
        ["pears", 900],
        ["bananas", 500]];
        const myObj = Object.fromEntries(fruits);
```

ii) optional catch binding(2019)

```
try {
        // code
} catch (err) {
        // code
    }
```

iii) Asynchronous iteration(2018)

```
let myPromise = new
promise()
myPromise.then();
myPromise.catch();
myPromise.finally();
```

v) JS object rest property(spread operator)(2018)

→This allows us to destruct an object and collect the leftovers onto a new object.

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 }; x; // 1
                y; // 2
                z; // { a: 3, b: 4 }
```

Vi) Nullish Coalescing(??)

(a>b)?a : B is:
if a is greater, then a,
else it is b.

Vii) Destructuring

```
Let person = {name: 'john', age: 21, gender: 'male'}

let { name, age, gender } =
person;

console.Log(name,age,
gender);
```

ix) Clouser

→In JavaScript, a closure is created when a function is defined inside another function and has access to the outer function's variables and parameters, even after the outer function has returned.

→In other words, a closure is a function with preserved data. It allows a function to access and manipulate variables that are outside of its own scope. The closure "closes over" these variables, keeping them in memory for as long as the closure is still in use.

Here is an example:

```
function outerFunction() {
  const outerVariable = "I am outside!";

  function innerFunction() {
    console.log(outerVariable);
  }

  return innerFunction;
}

const innerFunc = outerFunction();
innerFunc(); // Output: "I am outside!"
```

x) optional chaining

- **Optional chaining: (?.)**
  the optional chaining operator (?.) enables you to read the value of a property located deep within a chain of connected objects without having to check that each reference in the chain is valid.

  The ?. Operator is like the . Chaining operator, except that instead of causing an

error if a reference is nullish (null or undefined), the expression returns a value of undefined.

- Let x = foo?. Bar();
  if (foo?. Bar?. Baz) { // ... }

f) Browser Object Model(BOM)

i) js window

- The window object is supported by all browsers. It represents the browser's window.

  All global JavaScript objects, functions, and variables automatically become members of the window object.

  Global variables are properties of the window object.

  Global functions are methods of the window object.Even the document object (of the HTML DOM) is a property of the window object

- window.document.getElementById("header"); **is same as**

  document.getElementById("header");

- Window size

  window.innerHeight

  window.innerWidth

  window.open()

  window.close()

  window.moveTo()

  window.resizeTo()

ii) JS location

- window.location.href returns the href (URL) of the current page
- window.location.hostname returns the domain name of the web host
- window.location.pathname returns the path and

filename of the current page

- window.location.protocol returns the web protocol used (http: or https:)
- window.location.assign() loads a new document

iii)window extra

- alert() //Used to alert something on the screen
- blur() //The blur() method removes focus from the current window.
- setInterval(() => { //Keeps executing code at a certain interval

  // Code to be executed

  }, 1000);
- setTimeout(() => { //Executes the code after a certain interval of time

  // Code to be executed

  }, 1000);
- var name = prompt("What is your name?", "xyz");

  //Prompts the user with a text and

  takes a value. Second parameter is

  the default value
- window.scrollBy(100, 0); // Scroll 100px to the right
- window.scrollTo(500, 0); // Scroll to horizontal position 500
- clearInterval(var) //Clears the setInterval. var is the value returned by setInterval call
- clearTimeout(var) //Clears the setTimeout. var is the value returned by setTimeout call

- stop(). //Stops the further resource loading

iv) Cookies

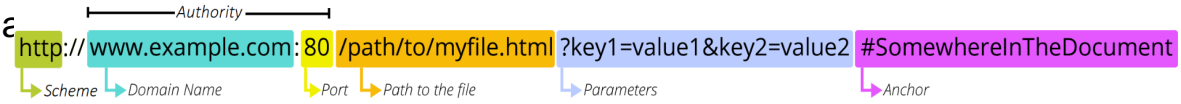
- Cookies are data, stored in small text files, on your computer.
- When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.
- Cookies were invented to solve the problem "how to remember information about the user":
    - When a user visits a web page, his/her name can be stored in a cookie.
    - Next time the user visits the page, the cookie "remembers" his/her name.
    - Cookies are saved in name-value pairs like:
    - username = John Doe

- document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";
    - add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed
    - With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.
    - let x = document.cookie; cookies can be read like this
    - When deleting cookies just don't value and include above syntax as it is.
    - You can overwrite the same cookies by mentioning another value in the same cookies name.

4. How web works?

a) A user enters a URL into a browser (for example, Google.com. This request is passed to a domain name server.
b) The domain name server returns an IP address for the server that hosts the Website (for example, 68.178.157.132).
c) The browser requests the page from the Web server using the IP address specified by the domain name server.
d) The Web server returns the page to the IP address specified by the browser requesting the page. The page may also contain links to other files on the same server, such as images, which the browser will also request.
e) The browser collects all the information and displays to your computer in the form of Web page.

5. **URL basics?**
    a. A url (uniform resource locator) is a unique identifier used to locate a resource on the internet. It is also referred to as a web a



    b.

    c. **A scheme.** The scheme identifies the protocol to be used to access the resource on the Internet. It can be HTTP (without SSL) or HTTPS (with SSL).
    d. **A host.** The host name identifies the host that holds the resource. For example, www.example.com. A server provides services in the name of the host, but hosts and servers do not have a one-to-one mapping. Refer to Host names.
    Host names can also be followed by a **port number**. Refer to Port numbers. Well-known port numbers for a service are normally omitted from the URL. Most servers use the well-known port numbers for HTTP and HTTPS , so most HTTP URLs omit the port number.
    e. **A path.** The path identifies the specific resource in the host that the web client wants to access. For example, /software/htp/cics/index.html.
    f. **A query string.** If a query string is used, it follows the path component, and provides a string of information that the resource can use for some purpose (for example, as parameters for a search or as data to be processed). The query string is usually a string of name and value pairs

6. **HTTP Format and Basics**
    - Hypertext transfer protocol (http) is a method for encoding and transporting information between a client (such as a web browser) and a web server. Http is the primary protocol for transmission of information across the internet.
    - Http is a tcp/ip based communication protocol, that is used to deliver data (html files, image files, query results, etc.) on the world wide web. The default port is tcp 80, but other ports can

be used as well. It provides a standardized way for computers to communicate with each other.

- Methods:
  - Get: read
  - Post: create
  - Patch: update
  - Delete: delete
  - Put: update/replace

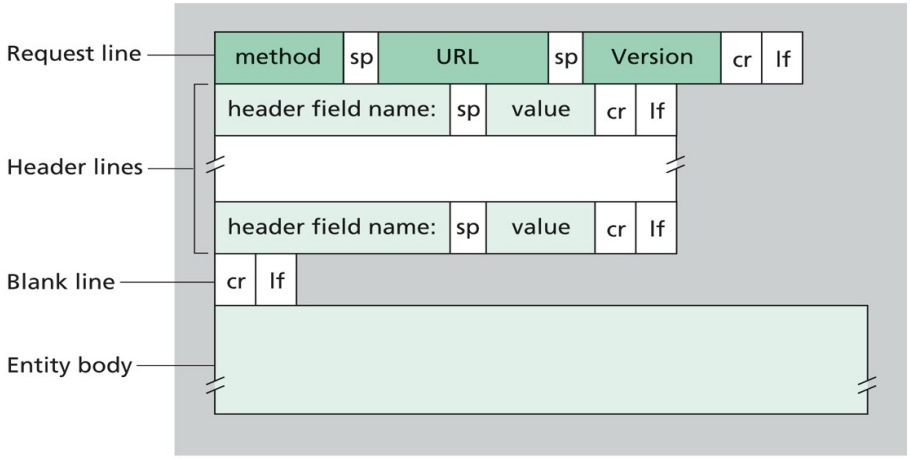


Figure 2.8 ♦ General format of a request message

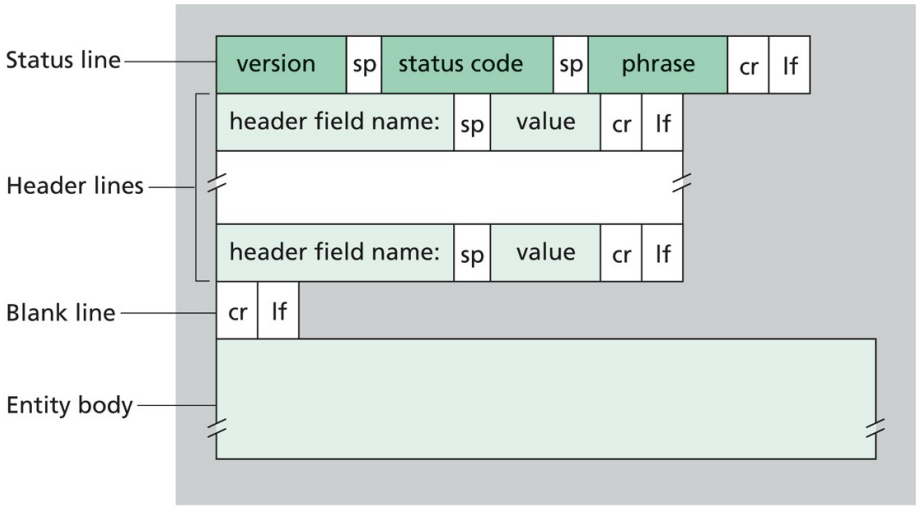


Figure 2.9 ♦ General format of a response message

7. Javascript interpreter or compiler?

a. Javascript is an interpreted language, not a compiled language. A program such as c++ or java needs to be compiled before it is run. The source code is passed through a program called a compiler, which translates it into bytecode that the machine understands and can execute.

8. **How javascript code gets executed?**

- Javascript is a synchronous (moves to the next line only when the execution of the current line is completed) and single-threaded (executes one command at a time in a specific order one after another serially) language. To know behind the scene of how javascript code gets executed internally, we have to know something called execution context and its role in the execution of javascript code.

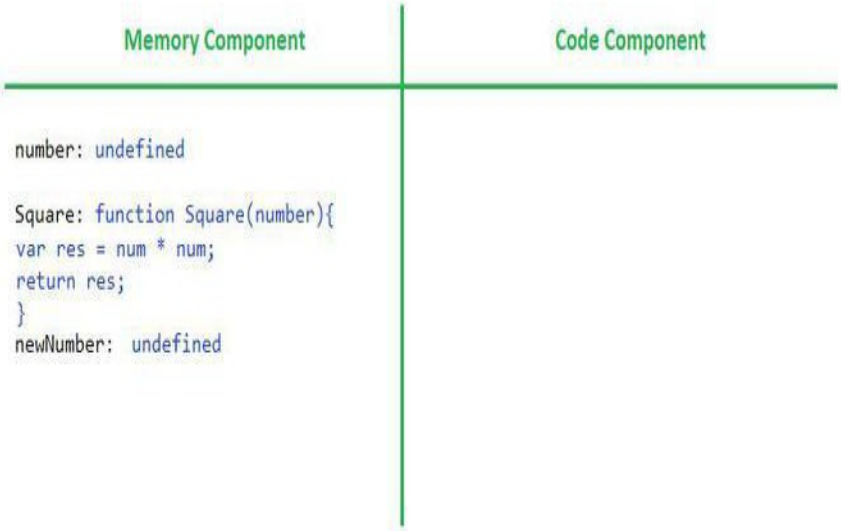
- Execution context: everything in javascript is wrapped inside execution context, which is an abstract concept (can be treated as a container) that holds the whole information about the environment within which the current javascript code is being executed.

- Now, an execution context has two components and javascript code gets executed in two phases.

- **Memory allocation phase:** in this phase, all the functions and variables of the javascript code get stored as a key-value pair inside the memory component of the execution context. In the case of a function, javascript copied the whole function into the memory block but in the case of variables, it assigns undefined as a placeholder.

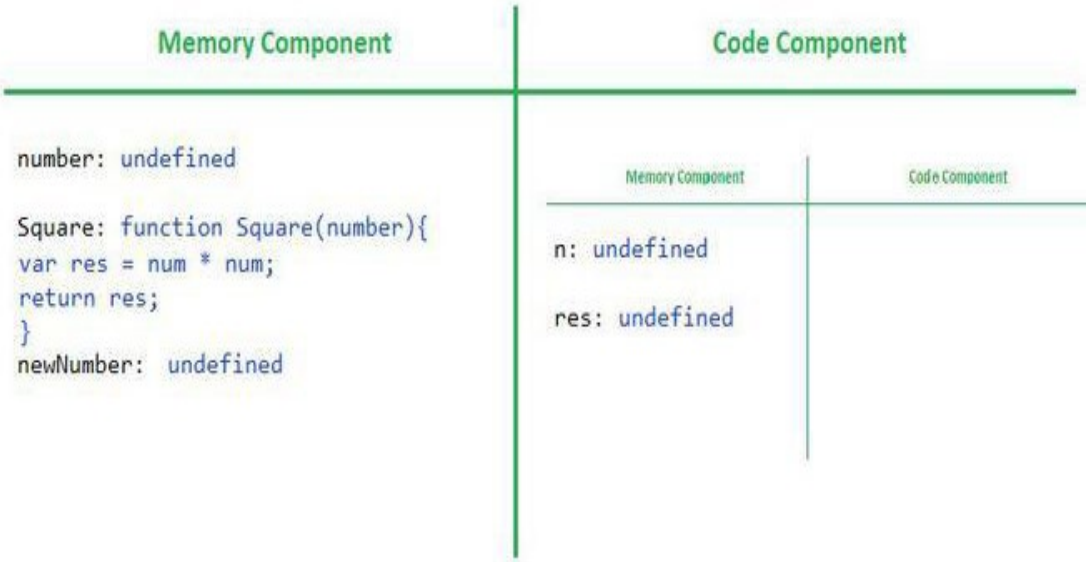
- **Code execution phase:** in this phase, the javascript code is executed one line at a time inside the code component (also known as the thread of execution) of execution context.

- Var number = 2;
  function square
  (n) { var res = n *
  n; return res;
  }
  var newnumber = square(3);
  So, in the memory allocation phase, the memory will be allocated for these variables and functions like this.

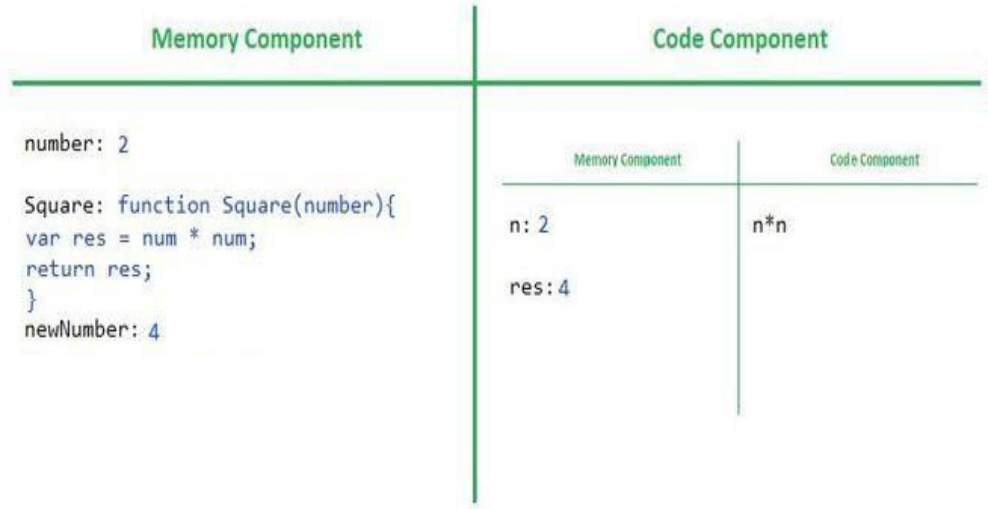


- In the code execution phase, javascript being a single thread language again runs through the code line by line and updates the values of function and variables which are stored in the memory allocation phase in the memory component.

- So in the code execution phase, whenever a new function is called, a new execution context is created. So, every time a function is invoked in the code component, a new execution context is created inside the previous global execution context.

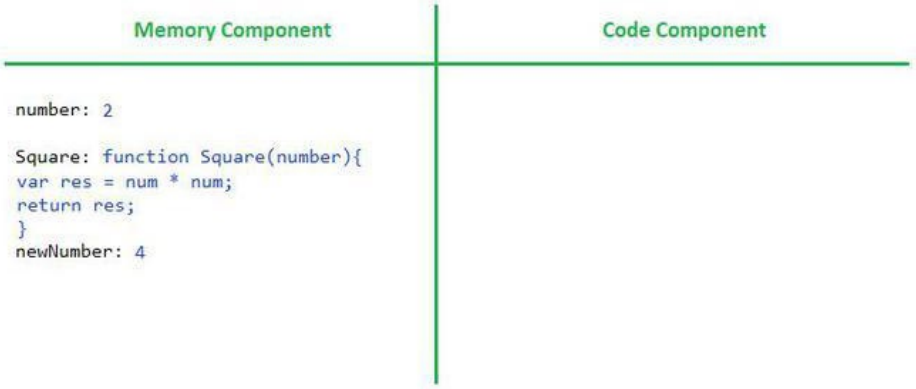


- So again, before the memory allocation is completed in the memory component of the new execution context. Then, in the code execution phase of the newly created execution context, the global execution context will look like the following.



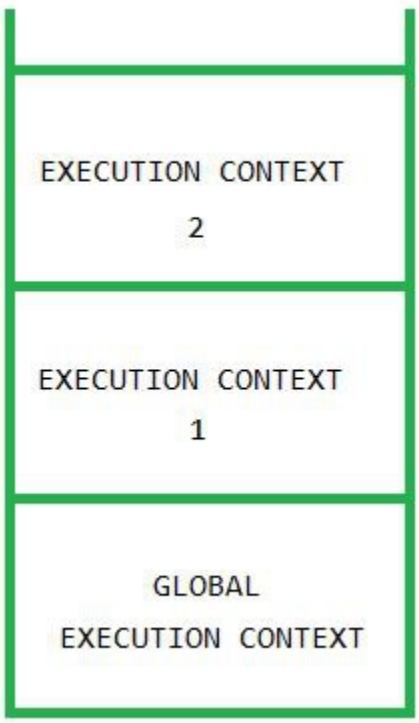
  - As we can see, the values are assigned in the memory component after executing the code line by line, i.e. number: 2, res: 4, newnumber: 4.

  - After the return statement of the invoked function, the returned value is assigned in place of undefined in the memory allocation of the previous execution context. After returning the value, the new execution context (temporary) gets completely deleted. Whenever the execution encounters the return statement, it gives the control back to the execution context where the function was invoked.



- After executing the first function call when we call the function again, javascript creates again another temporary context where the same procedure repeats accordingly (memory execution and code execution). In the end, the global execution context gets deleted just like child execution contexts. The whole execution context for the instance of that function will be deleted

- call stack: when a program starts execution javascript pushes the whole program as global context into a stack which is known as call stack and continues execution. Whenever javascript executes a new context and just follows the same process and pushes to the stack. When the context finishes, javascript just pops the top of the stack accordingly.



- When javascript completes the execution of the entire code, the global execution context gets deleted and popped out from the call stack making the call stack empty.

9. **Interview Question**
    a. https://www.guru99.com/javascript-interview-questions-answers.html
    b. https://www.javatpoint.com/javascript-interview-questions
    c. **https://www.geeksforgeeks.org/javascript-interview-questions-and-answers/**